PCI Win32 Driver Software
User's Manual

ACROMAG INCORPORATED
30765 South Wixom Road
P.O. BOX 437
Wixom, MI 48393-7037 U.S.A.

Tel: (248) 295-0310
Fax: (248) 624-9234

9500-280K

Contents

Introduction

PCI Win32 Driver Software (PCISW-API-WIN) consists of low-level drivers and Windows 32 Dynamic Link Libraries (DLLs) that facilitate the development of Windows applications accessing Acromag PMC I/O module products (e.g. PMC730), PCI I/O Cards (e.g. APC730) and CompactPCI I/O Cards (e.g. AcPC730).

Note:
The convention of this document is to refer to the above hardware devices using the generic name "PCI modules."

DLL functions use the Windows _stdcall calling convention and can be accessed from a number of programming languages. In addition to the DLLs and drivers, the software includes an ActiveX control for implementing interrupt notifications in programming environments that do not support the use of callback functions. Several example C, Visual Basic and LabVIEW applications are also provided with source code.

This document covers general information on software installation, programming concepts, application development and redistribution issues. Also included in the PCI Win32 Driver documentation is a set of Function Reference documents for each Acromag PCI module DLL. After reviewing this user's manual, readers will next want to consult the Function Reference document specific to their hardware.

Hardware Support

The list of supported PCI devices is shown in Table1.

Table 1: Acromag PCI Modules

Model	Description	Interrupts
PCI220	16 Non-Isolated 12-Bit DAC Outputs	No
PCI230	8 Ch., 16-Bit DAC Outputs	No
PCI330	16 Bit (16DE/32SE) Analog Input Module	Yes
PCI341	12/14 Bit (16DE) Simultaneous Analog Input Module	Yes
PCI408	32 Non-Iso. Digital Inputs/Outputs (Low Side Sw.)	Yes
PCI424	Differential I/O Counter Timer Module	Yes
PCI440	32 Ch., Isolated Digital Input Module with Interrupts	Yes
PCI464	Digital I/O Counter Timer Module	Yes
PCI470	48 Ch., Digital I/O Module with Interrupts	Yes
PCI482	10 16-bit counters – TTL	Yes
PCI483	4 16-bit counters – TTL and 4 32-bit counters – RS422	Yes
PCI484	6 32-bit counters – RS422	Yes
PCI520	8 Channel, Serial 232 Communication	Yes
PCI521	8 Channel, Serial 422/485 Communication	Yes
PCI730	Multi-Function Input/Output Board	Yes
PCIAX1020	Reconfigurable, 1M gates, Four 20MHz A/D, Two 900 KHz D/A	Yes
PCIAX1065	Reconfigurable, 1M gates, Four 64MHz A/D, Two 900 KHz D/A	Yes
PCIAX2020	Reconfigurable, 2M gates, Four 20MHz A/D, Two 900 KHz D/A	Yes
PCIAX2065	Reconfigurable, 2M gates, Four 64MHz A/D, Two 900 KHz D/A	Yes
PCIAX3020	Reconfigurable, 3M gates, Four 20MHz A/D, Two 900 KHz D/A	Yes
PCIAX3065	Reconfigurable, 3M gates, Four 64MHz A/D, Two 900 KHz D/A	Yes
PCICX1002	Reconfigurable, 32 differential I/O; 11,500 LC	Yes
PCICX1003	Reconfigurable, 16 TTL and 24 differential I/O; 11,500 LC	Yes
PCICX2002	Reconfigurable, 32 differential I/O; 24,192 LC	Yes
PCICX2003	Reconfigurable, 16 TTL and 24 differential I/O; 24,192 LC	Yes
PCIDX501	Reconfigurable, 500K gates, 64 TTL I/O	Yes
PCIDX502	Reconfigurable, 500K gates, 32 Differential I/O	Yes
PCIDX503	Reconfigurable, 500K gates, 24 Differential, 16 TTL I/O	Yes
PCIDX2001	Reconfigurable, 2M gates, 64 TTL I/O	Yes
PCIDX2002	Reconfigurable, 2M gates, 32 Differential I/O	Yes
PCIDX2003	Reconfigurable, 2M gates, 24 Differential, 16 TTL I/O	Yes
PCILX40	Reconfigurable, 41,472 LC, I/O Board	Yes
PCILX60	Reconfigurable, 59,904 LC, I/O Board	Yes
PCISX35	Reconfigurable, 34,560 LC, 192 DSP, I/O Board	Yes
PCIVFX70	Reconfigurable, 44,800 CLB FF, 128 DSP Slices, Power PC Core	Yes
PCIVLX85	Reconfigurable, 51,840 CLB Flip Flops, 64 DSP Slices	Yes
PCIVLX110	Reconfigurable, 69,120 CLB Flip Flops, 64 DSP Slices	Yes
PCIVLX155	Reconfigurable, 97,280 CLB Flip Flops, 640 DSP Slices	Yes
PCIVSX95	Reconfigurable, 58,880 CLB Flip Flops, 640 DSP Slices	Yes

Language Support

PCI Win32 Driver Software has been tested in the following development environments:

- Visual C++ 6.0, .NET 2003, 2005 and 2008
- Borland C++ Builder 5 and 6
- Visual Basic . NET 2003, 2005 and 2008
- National Instruments LabVIEW 6i and 7

Getting Started

Software Installation

To install the PCI Win32 Driver software, insert the PCI Win32 Driver disk into the CD drive and run **Setup.exe**. Note that administrative rights are required to perform the installation.

INSTALLED SOFTWARE

The default installation directory is `C:\Acromag\PCISW_API_WIN`.

Subdirectory	
c_examples	Microsoft Visual C++ and Borland C++ Builder examples with source code
c_include	Header files
c_lib	COFF format import libraries
config_files	Example VHDL object code for reconfigurable FPGA I/O Modules
docs	User's manual, DLL references, application notes
labview examples	LabVIEW 6i and 7 examples
redist	INF files, kernel driver, and DLLs
utility	PCIEnum utility
vb_examples	Visual Basic.NET examples
vb_include	Visual Basic function prototype modules

In addition, the PCI Event ActiveX control is installed to
`C:\Program Files\Common Files\Acromag\PCIEvent`
This file is shared by IPSW-API-WIN and IOSSW-API-WIN.

Hardware Installation

1. Configure any jumpers on the PCI modules as necessary.
2. With power off, install the PCI module into an available slot on the PC. Connect any field wiring at this time.
3. Turn on the PC. You will receive a dialog box shortly after boot-up asking if you want to install a driver for the new device. Answer yes and direct the New Hardware Wizard to either the "redist" subdirectory (see table above) or to the optical drive containing the PCI Win32 Driver disk. The New Hardware Wizard will copy and install the kernel mode driver.
   Note:
   - If the New Hardware Wizard is bypassed the driver can be installed from the Device Manager instead. Within Device Manager, locate and right-click the "PCI Data Acquisition and Signal Processing Controller," select Update Driver, and browse to the driver files as described above.

PCI Enumeration Utility

PCI Win32 Driver Software includes a command line utility, **PCIEnum.exe** which may be run to display basic information about all installed Acromag PCI modules. This information includes the module name, card number, bus number, device number, vendor ID, device ID, PCIBAR0 base address and Irq. The card number is the value passed to the `PCIXXX_Open` function to open a connection to the device. (See the **Sequence of Operations** section below.)

Software Overview

The software includes Windows 32 DLLs for each Acromag PCI module. In most cases the name of the DLL matches the name of the PCI module. For example the PMC470 is used with PCI470.dll. There are a few exceptions, however, where groups of similar PCI modules are supported by a single DLL. These include:

PCI Modules	Shared DLL
AX Series	PCIAX.dll
CX Series	PCICX.dll
DX Series	PCIDX.dll
LX and SX Series	PCILX.dll
VLX and VSX Series	PCIVLX.dll

The DLLs provide the Application Programming Interface (API) used to access the hardware. Each DLL is written in C and contains functions using the _stdcall calling convention. The DLL is loaded and linked at runtime when its functions are called by an executable application. Multiple applications can access the functions of a single copy of a DLL in memory.

In addition to the DLLs, the software also includes an ActiveX control that may be used to implement interrupt notifications in programming environments that do not support the use of callback functions.

Function Format

All PCI DLL functions have the following form:

```
status = PCIXXX_FunctionName(arg1, arg2, ... argn)
```

The "PCIXXX" portion of the function name indicates the PCI module the function is used with (e.g. PMC470).

Every function returns a 16-bit status value. This value is set to 0 when a function completes successfully or to a negative error code if a problem occurred. The following **Status Codes** section describes the values that may be returned from the DLL functions.

For most functions, *`arg1`* is an integer "handle" used to reference a specific PCI module. (See the **Sequence of Operations** section below.)

STATUS CODES

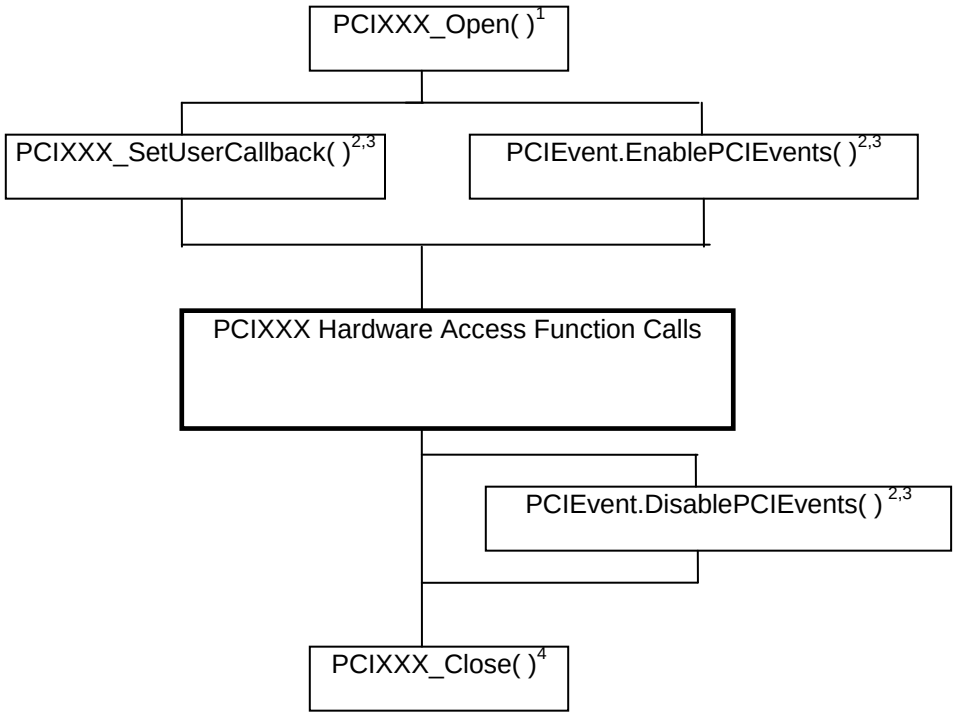
The table below summarizes the 16-bit status codes that may be returned from the DLL functions and ActiveX control methods. Please note the return code of any failing functions if you should need to contact Acromag for technical support.

Value	Mnemonic	Description
0	OK	Operation Successful
-1	ERR_INVALID_HNDL	Returned if no board is associated with the specified handle. Applies to most functions.
-2	ERR_CARD_IN_USE	Returned by *PCIXXX_Open()* if card is already open. This can occur if the card is in use by another application.
-3	ERR_NEWDEV	Returned by *PCIXXX_Open()* if an error occurred creating a software instance of the device
-4	ERR_CONNECT	Returned by *PCIXXX_Open()* if an error occurred connecting to the device. This will occur if the specified card number is invalid or if the kernel mode drivers are not properly installed or configured.
-5	ERR_MAPMEM	Returned by *PCIXXX_Open()* or *PCIXXX_Close()* if an error occurred mapping the devices physical memory into the virtural address space.
-6	ERR_THREAD	Returned by *PCIXXX_Open()* for devices that support interrupts if an error occurred while creating the interrupt service routine thread.
-7	ERR_ISR_ENABLE	Returned by *PCIXXX_Open()* for devices that support interrupts if an error occurred while enabling interrupt support for the board.
-8	ERR_OUTOFHANDLES	Returned by *PCIXXX_Open()* if an attempt is made to have more than 10 cards simultaneously open.
-9	ERR_BAD_PARAM	Returned by a function if it received an invalid parameter. This typically means the parameter was outside of the expected range or the function received a NULL pointer. Consult the individual function descriptions for other possible reasons for this error.
-10	ERR_INSUF_MEM	Returned by a function if there was insufficient memory to created a required data structure or carry out an operation.
-11	ERR_OCX_IN_USE	Returned by the ActiveX method *EnablePCIXXXEvents* if the control is already configured for use by another PCI module
-12	ERR_DLL_LOAD	Returned by ActiveX methods if the PCIXXX DLL can not be loaded
-13	ERR_CONFIG_READ	Returned by PCIXXX_*ReadPCIReg* if an error occurred while reading data from the device's PCI configuration space.
-14	ERR_TIMEOUT	Returned by a function if it timed out before completing.
-15	ERR_CONFIG_SET	Returned by a Configuration function if the current settings used by this function do not represent a valid configuration
-16	ERR_CALIB	Indicates an error generating or using calibration data.
-17	ERR_BUFFER	Indicates an error occurred accessing a user defined data buffer
-18	ERR_CONFIG_WRITE	Returned by PCIXXX_*WritePCIReg* if an error occurred while writing data to the device's PCI configuration space.

-19	ERR_DMA_MAP	Returned by a DMA function if a DMA buffer is not mapped into the process
-20	ERR_EEPROM_ACK	Returned by EEPROM access functions if an Acknowledge is expected but not received
-21	ERR_EEPROM_ READBACK	Returned by EEPROM write if the value read from the EEPROM does not match value written
-22	ERR_FILE_OPEN	Returned if a file cannot be opened
-23	ERR_FILE_FORMAT	Returned if file contents are not in the expected format
-24	ERR_FILE_READ	Returned if a file cannot be read
-25	ERR_CONFIG_DONE	Returned by functions if the configuration of a re-configurable board is not complete
-26	ERR_EX_DESIGN	Returned by functions if a re-configurable board is not configured with the Acromag supplied example design
-27	ERR_HARDWARE	Returned if a hardware malfunction is detected
-28	ERR_FLASH_BUSY	Returned if a flash operation cannot be carried out because the flash chip is busy
-29	ERR_UNSUPPORTED	Returned if the device does not support the requested action
-30	ERR_CHECKSUM	Returned if a checksum mismatch is detected
-31	ERR_HANDLER	Returned if the function requires an interrupt handler and one has not yet been attached..

Sequence of Operations

Although each PCI module has its own DLL with unique functions, all PCI modules are accessed using the calling sequence shown below.



Notes:

1. PCIXXX_Open provides an integer "handle" used to access the specific PCIXXX module in all subsequent calls.
2. Not all Acromag PCI modules support interrupts. User callbacks or ActiveX event handlers are used to implement custom interrupt service routine logic for reconfigurable boards.
3. User callbacks and ActiveX event notifications are mutually exclusive.
4. PCIXXX_Close should always be called for each "open" PCI module prior to terminating the application.

Interrupts

PCI Win32 Driver Software provides two mechanisms for allowing your application to respond to interrupts generated in the hardware: callback functions and ActiveX event notifications. An application may implement one or the other but not both. In general, the mechanism you use will be dictated by your choice of programming language. Callback functions are the preferred technique due to lower latency, but they are not fully supported by all development environments.

Each PCI DLL that supports interrupts has its own predefined internal interrupt service routine. (DLLs for reconfigurable boards are a special case. See note below.) The specifics of each routine are outlined in the PCI module's corresponding Function Reference document. If you choose to implement a callback function or use ActiveX event notification, you have the option of overriding this routine. This is done by setting a "*Replace*" parameter when designating the callback function or during configuration of the ActiveX control. (See Callback Functions and PCI Event ActiveX Control)

When an interrupt occurs the following sequence of events takes place:

1. The kernel level driver disables the board's Interrupt Enable bit and signals the DLL's internal interrupt service routine (ISR).
2. At this point three things can happen
    a. If no callback or event notification was configured, the ISR simply processes the interrupt and then re-enables the board's Interrupt Enable bit.
    b. If a callback function or event notification was configured but should not override the internal ISR, the internal ISR processes the interrupt, re-enables the board's Interrupt Enable bit and then either invokes the callback or notifies the ActiveX control to fire an event.
    c. If a callback function or event notification was configured to override the internal ISR, the ISR invokes the callback or notifies the ActiveX control to fire an event and then immediately returns without further processing. It is then the responsibility of the callback function or event handler to process the interrupt and re-enable the Interrupt Enable bit.

Note
The DLLs for user programmable FPGA boards include only a bare-bones ISR. When using interrupts, applications for these boards must include a callback function or event handler to implement the custom ISR logic. In other word, overriding the internal ISR is mandatory. (The PCIAX and PCIDX DLLs are exceptions to this rule in that they do have "complete" internal ISRs. However, these ISRs are specific to the provided example FPGA designs. If custom FPGA logic is used, the internal ISR must be overridden with a callback function or event handler.)

CALLBACK FUNCTIONS

Callback functions are supported in C/C++ and Visual Basic .NET.

When using the callback mechanism your application defines a function that the DLL will call from its internal interrupt service routine. The format of this function must exactly match that expected by the DLL. This format is hardware specific and is given in the **PCIXXX_SetUserCallback** topic in the PCI module's Function Reference document.

This format, however, will be some variation of the following:

```
C:   void (_stdcall *ISR)(short Handle, WORD Status)
VB:  Sub ISR(ByVal Handle As Short, ByVal Status As Short)
```

The *Handle* argument identifies the board that caused the interrupt. If the function is not overriding the internal ISR, the Status variable(s) will contain data allowing you to determine the cause of the interrupt (e.g. the value read from a status register by the internal ISR). If the function is overriding the internal ISR, the Status variable(s) will be zero since the internal ISR did not read any registers prior to invoking the callback function.

PCI EVENT ACTIVEX CONTROL

The PCI Event ActiveX control (**PCIEvent.ocx**) may be used for interrupt notification and processing in environments that do not support callback functions (LabVIEW) or where there are complications implementing thread safe code.

Note**:** Although the ActiveX control may also be used in Visual Basic .NET and windowed C/C++ applications, the callback approach is recommended due to its lower latency.

Instructions on adding the ActiveX control to an application and defining event handlers is deferred until the LabVIEW topic of the **Building Windows Applications** section. The general rules for using the control, however, are as follows:

1. The PCI Event control can only be associated with one board at a time. Add one control to the application for each PCI module you wish to receive interrupt notifications from.
2. To associate the control with a board, call the ActiveX method *EnablePCIEvents* passing the board handle received from *PCIXXX_Open* and the board's device ID. Set the method's *Replace* parameter to indicate whether your event handler should override the DLL's internal ISR.
3. In many cases the PCI Event control can fire two types of events for the same interrupt condition. One type will pass argument(s), such as the value of a status register, which can be inspected to determine the interrupt source. The other type of event does not pass any arguments. The source of the interrupt can be determined from the event name.

   For example, if interrupt conditions are sensed on input channels 0 and 6 of a PMC408, the control will fire *Bit0*, *Bit6* and *PCIEvent1w(0x41)*.

   Consult the DLL's Function Reference document to determine which events can be fired for your hardware. In general, applications will include just one of the handlers for a given interrupt condition. Which handlers you choose to implement will depend on the nature of your application.
4. Call *DisablePCIEvents* to disassociate the control from a board prior to calling *PCIXXX_Close* to close the board.

SYNCHRONIZATION

The DLL's interrupt service routine (ISR) executes on a different thread than that of your application. Within the DLL the ISR (which includes the call to any callback function) is delimited as a device critical section. *PCIXXX_StartIsrSynch* and *PCIXXX_EndIsrSynch* can be used to synchronize other application threads with the ISR thread and to synchronize multiple threads within an application with each other.

Bracketing a section of code between calls of *PCIXXX_StartIsrSynch* and *PCIXXX_EndIsrSynch* defines that code as a device critical section. Two threads within a single process cannot execute critical section code simultaneously. *PCIXXX_StartIsrSynch* should be called by your application before it attempts to access data or device memory that can be accessed by another thread. Remember to call *PCIXXX_EndIsrSynch* when finished accessing these shared resources.

Code in an ActiveX event handler function is not automatically defined as a critical section. If desired, *PCIXXX_StartIsrSynch* and *PCIXXX_EndIsrSynch* may be used to bracket this code and synchronize its execution with your application.

BASE ADDRESS POINTERS

Each DLL provides a function that returns the base address of the user mode mapping of the PCI module's I/O space.

C and C++ programmers can cast the returned value to a byte pointer and access memory using normal pointer mechanisms. This method can be used to write additional functions that complement those provided through the DLL.

Example

```
/* Read PMC408 Digital Input Channel Register A */

DWORD base_address;
volatile BYTE* pbase_addr;
WORD chan_val;

if (PCI408_GetBaseAddress(Handle, &base_address) == 0)
{
   pbase_addr = (BYTE*)base_address;
   chan_val = *(PWORD)(pbase_addr + 0x200);
}
```

Building Windows Applications

This section describes the basic steps to create applications that use the PCI Win32 dynamic link libraries and ActiveX control.

Steps are outlined for building applications in the following development environments:

- Microsoft Visual C++ 6 (VC6)
- Microsoft Visual C++.NET
- Borland C++ Builder
- Visual Basic .NET
- LabVIEW 6i and 7

C/C++

MICROSOFT VISUAL C++ 6

The steps to create a C or C++ application using VC6 are as follows:

1. Open a new or existing Visual C++ project.
2. Add the path to the necessary header files (PCIXXX.h, PCIErrorCodes.h) to the project's **Preprocessor | Include** directories setting. This is located under **Project | Settings | C/C++**.
3. Add the path to the import library (PCIXXX.lib) to the project.
   To do this:
   - Open **Project | Settings | Link**
   - Select the **Input** category and modify the **Additional Library Path** field
   - Add the import library name to the **Object/library modules** field
4. Include the windows.h, PCIXXX.h and PCIErrorCodes.h header files at the beginning of your .c (C source code) or .cpp (C++ source code) files.
   e.g.
   ```
   #include <windows.h>
   #include "PCI408.h"
   #include "PCIErrorCodes.h"
   ```
5. Build your application

MICROSOFT VISUAL C++ .NET

The steps to create a C or C++ application using VC.NET are as follows:

1. Open a new or existing Visual C++ project.
2. Add the path to the necessary header files (PCIXXX.h, PCIErrorCodes.h) to the project. To do this, open the project's property pages, open the **C/C++** folder, select the **General** property page and modify the **Additional Include Directories** property.
3. Add the path to the import library (PCIXXX.lib) to the project.
   To do this:
   - Open the project's property pages
   - Open the **Linker** folder, select the **General** property page and modify the **Additional Library Directories** property
   - Select the **Input** property page and add the import library to the **Additional Dependencies** property
4. Include the windows.h, PCIXXX.h and PCIErrorCodes.h header files at the beginning of your .c (C source code) or .cpp (C++ source code) files.
   e.g.
   ```
   #include <windows.h>
   #include "PCI408.h"
   #include "PCIErrorCodes.h"
   ```
5. Build your application

BORLAND C++ BUILDER

The steps to create a C or C++ application using C++ Builder 5 and 6 are as follows:

1. Open a new or existing C++ Builder project.
2. Add the path to the necessary header files (PCIXXX.h, PCIErrorCodes.h) to the project's **Include path** setting. This is located under **Project | Options | Directories/Conditionals**.
3. The provided import library (PCIXXX.lib) uses Microsoft's COFF format. Since the COFF format is not compatible with Borland's OMF format it is necessary to create a compatible import library using Borland's IMPLIB utility.

   IMPLIB works like this: `IMPLIB (destination lib name) (source dll)`
   For example: `IMPLIB OMF_PCI408.lib PCI408.dll`
4. Add the new import library to the project by selecting **Project | Add to Project…** and browsing to the lib file.
5. Include the windows.h, PCIXXX.h and PCIErrorCodes.h header files at the beginning of your .c (C source code) or .cpp (C++ source code) files.
   e.g.
   ```
   #include <windows.h>
   #include "PCI408.h"
   #include "PCIErrorCodes.h"
   ```
6. Build your application

Visual Basic

VISUAL BASIC .NET

The steps to create an application using Visual Basic .NET are as follows:

1. Open a new or existing project.
2. Add the files containing the DLL function prototypes (PCIXXX.vb) and the error code constants (PCIErrorCodes.vb) to the project. To do this, select **Project | Add Existing Item** from the menu and select the desired files.

The following steps are necessary if you will be implementing a callback function. If you will not be using a callback skip to step 9.

In Visual Basic .NET callback functions are implemented using delegates. A delegate is a class that can hold a reference to a method and is equivalent to a type-safe function pointer or a callback function.

3. Add a new module to the project that will contain the callback function. To do this, select **Project | Add Module** from the menu, select the **Module** template and select **Open**.
4. Add an ISR subroutine to the new module. The format of the routine is hardware specific and is given in the PCIXXX_SetUserCallback topic in the PCI module's Function Reference document.

   This format, however, will be some variation of the following:

```
Sub ISR(ByVal Handle As Short, ByVal Status As Short)

End Sub
```

5. In the declares section of your form code declare a garbage collection handle:

```
Dim gch As GCHandle
```

   (Note: If the editor indicates *GCHandle* is an undefined type, add `Imports System.Runtime.InteropServices` to the top of the source file.)

   The DLL will store the delegate passed to it for later use. Since the DLL is unmanaged code, it is necessary to manually prevent garbage collection of the delegate until the DLL is through with it.
6. Include the following statements prior to the call to PCIXXX_SetUserCallback:

```
Dim dlg As ISRDelegate     'ISRDelegate is defined in PCIXXX.vb
dlg = AddressOf ISR        'assign delegate to callback function
gch = GCHandle.Alloc(dlg)  'protect the delegate from garbage
                           'collection.
```

7. Now notify the DLL that it should invoke the callback

```
PCIXXX_SetUserCallback (pciHandle, dlg, fReplace)
```

8. At the end of your program, remember to free the garbage collection handle:

```
PCIXXX_Close(pciHandle)
gch.Free()
```

9. Build your application.

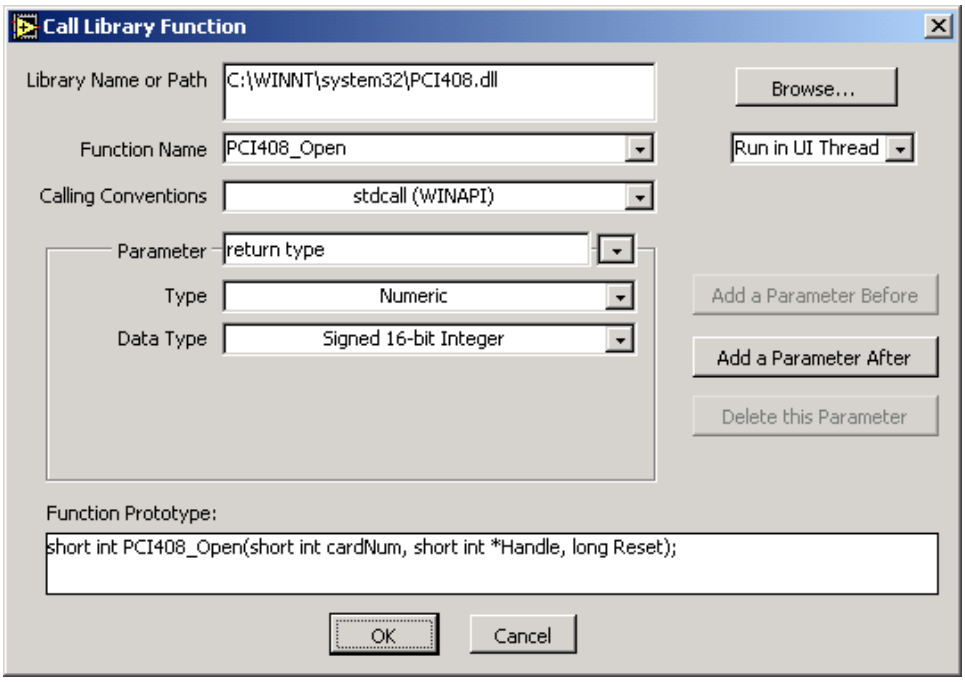
LabVIEW

The steps to create an application using LabVIEW 6i or 7 are as follows:

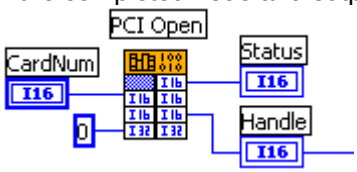
1. Open a new or existing VI.

   In LabVIEW DLL functions are called using Call Library Function nodes. The following steps outline how to add and configure one of these nodes.

2. Select **Call Library Function** from the **Functions | Advanced** subpalette and then click within the block diagram.
3. Right-click on the node and select **Configure**. This opens the Call Library Function configuration dialog.
4. Click the Browse button and locate the desired DLL within the Windows/System, WINNT/system32 or Windows/system32 directory. When finished, the name of the DLL will be displayed in the **Library Name or Path** field. Setting the Library Name in this manner automatically populates the **Function Name** Combobox.
5. Select the desired function from the **Function Name** Combobox.
6. Set the **Calling Conventions** to stdcall(WINAPI).
7. Leave the menu below the **Browse** button set to **Run in UI Thread** unless you plan on using the IsrSynch functions to make your DLL function calls thread safe.
8. Configure the return type as Numeric, Signed 16-bit Integer.
9. Add and configure additional parameters until the Function Prototype field matches the LabVIEW syntax listed for the function in the DLL's Function Reference guide. An example of a finished Call Library Function configuration dialog is shown below.



10. Wire inputs to the left side of the completed node and outputs to the right.



The following steps show how to use the PCI Event ActiveX control for event notifications in the VI.

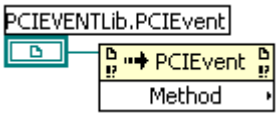
Adding the ActiveX control to a VI

1. LV6i: Select Container from the **Controls | ActiveX** subpallete and then click a point within the front panel to add the control to the panel.
   LV7: Select ActiveX Container from the **All Controls | Containers** subpallete and then click a point within the front panel to add the control to the panel.
2. Right click within the container and select the PCIEvent control. After clicking **OK** the control will appear as a white box within the front panel. In addition, a node labeled "PCIEVENTLib.PCIEvent" will appear in the block diagram window.

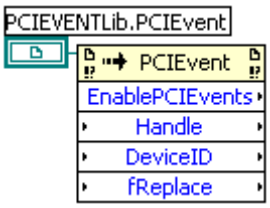
Invoking an ActiveX method

The methods, *EnablePCIEvents* and *DisablePCIEvents* are used to associate and disassociate the control from the handle (and hardware) used with the DLL functions. ActiveX methods are invoked as follows:

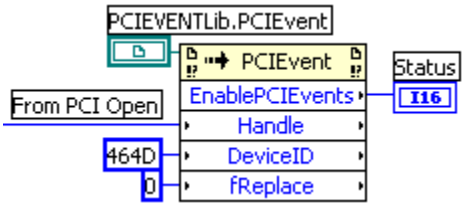
1. Select **Invoke Node** from the **Functions | Communication | ActiveX** subpalette and then click within the block diagram.
2. Wire the ActiveX control added previously to the **refnum** input on the node as shown below.



3. Right click on "Methods" to display a pop-up menu and select the desired method from the **Methods** sub-menu. The node will now show this method along with entries for its parameters.

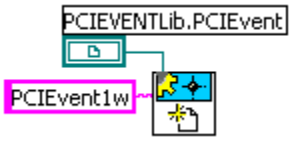


4. Wire parameters (constants, type compatible controls and variables) to the left side of the node. Return values are wired to right of the method name.



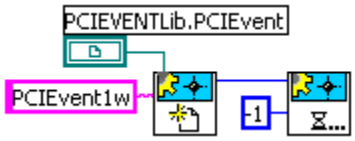
Working with ActiveX Events (LabVIEW 6i)
The PCI Event ActiveX control fires events when an interrupt occurs in the control's corresponding PCI module. This section outlines how to monitor and respond to ActiveX events within the VI.

1. Select **Create ActiveX Event Queue.vi** from the **Functions | Communication | ActiveX | ActiveX Events** subpalette and then click within the block diagram.
2. Wire the previously added ActiveX control to the **refnum** input on the node.
3. Locate and right click on the **Event Name** terminal. Select **Create Constant** and type in the name of the event to monitor. This name can be obtained from the PCI module's Function Reference document. The name is case sensitive. The diagram should look as follows:

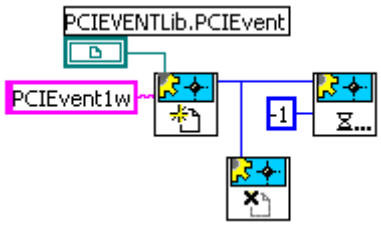


4. Select **Wait On ActiveX Event.vi** from the **ActiveX Events** subpalette and add the VI to the diagram. Wire the **Event Queue** terminals of the two VIs together. Locate the **ms timeout (-1)** terminal and add a constant to indicate the number of milliseconds the VI will wait for the event before timing out. A value of –1 means wait indefinitely.

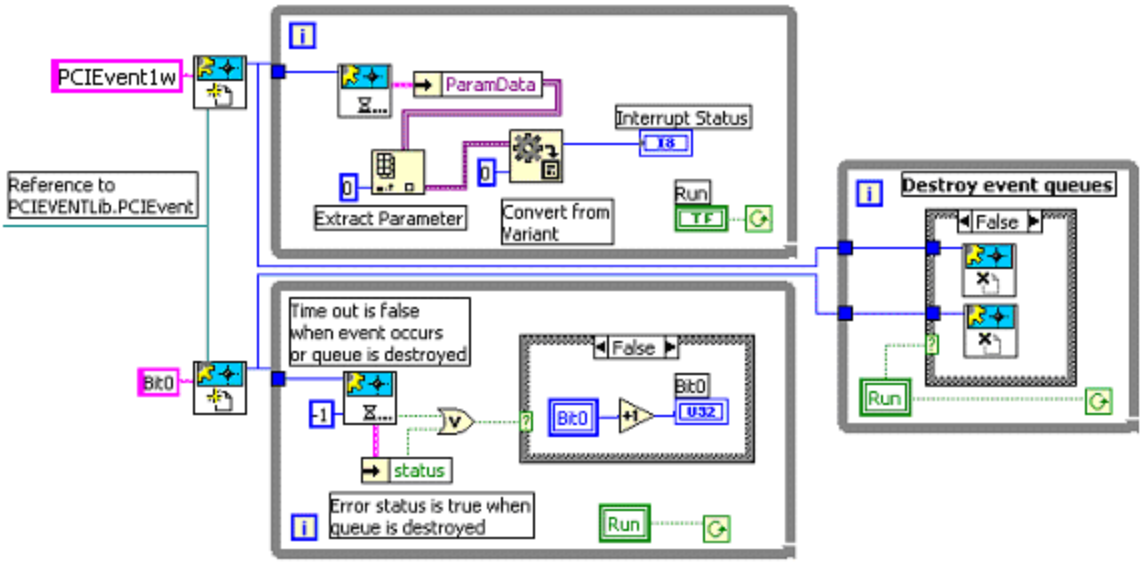
   The output wiring from the VI will depend on if the selected event passes an argument or not. If the event passes arguments, wire the Event Data terminal, otherwise, wire the timed out terminal. (See complete example at the end of this section.)



5. Select **Destroy ActiveX Event Queue.vi** for the ActiveX Events subpalette and add the VI to the diagram. Wire the VI's **Event Queue** terminal to the **Event Queue (out)** terminal of the **Wait On ActiveX Event** VI. In the completed example, a control structure is used to determine when this VI executes. (See the complete example at the end of this section.)



6. Below is a complete block diagram showing the processing of events that do (PCIEvent1w) and do not (Bit0) pass arguments.



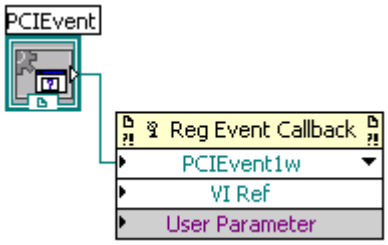
In the example above, the two Wait On ActiveX Event while loops execute as long as the Run button is activated. When the button is deactivated, both event queues are destroyed.

The upper **Wait On ActiveX Event** VI is used to monitor an event that returns a status variable. When an event occurs the VI returns the event data as a cluster. The parameter data and names are then unbundled as an array. Next, the parameter data is extracted from the array and converted from variant to integer data. Finally the status value is displayed in an indicator on the front panel.

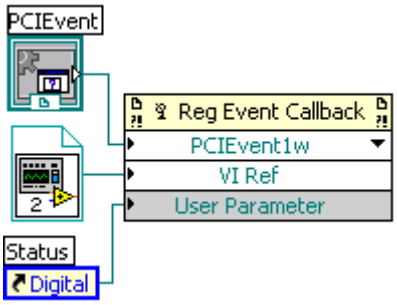
The lower **Wait On ActiveX Event** VI is used to monitor Bit0 events. The VI's **timed out** terminal returns false when an event occurs or the event queue is destroyed. If the error status is also false, a Bit0 event indicator is incremented on the front panel. (The error status is set to true when the event queue is destroyed.)

Working with ActiveX Events (LabVIEW 7)
The PCI Event ActiveX control fires events when an interrupt occurs in the control's corresponding PCI module. This section outlines how to register a VI to be called when a specific ActiveX event occurs.

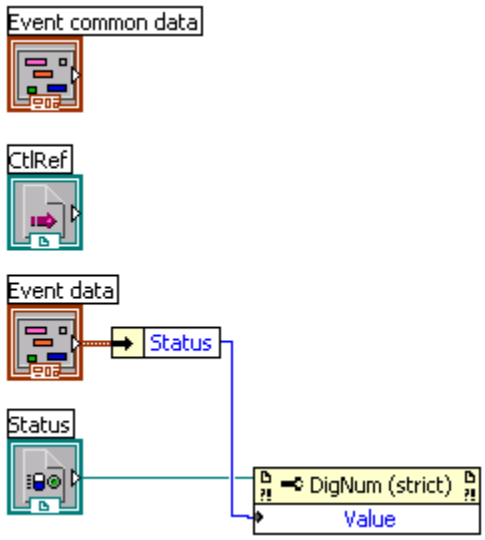
1. Select Register Event Callback from the **Functions | Communication | ActiveX** subpallete and then click within the block diagram.
2. Wire the previously added ActiveX control to the **event source ref** input on the node.
3. Click the down arrow next to the event source ref input and select the ActiveX event to be handled. The diagram should look as follows:



4. The User Parameter input can be used to transfer data between the callback VI and the main VI. For example, if a reference to an indicator on the front panel of the main VI is wired to this input, the callback VI can update the indicator each time the specified event occurs.
5. Right-click on the VI Ref portion of the node and select **Create Callback VI** from the context menu. At this point registration of the callback is complete. If necessary, the node can be resized to register callbacks for additional events.



6. The callback VI LabVIEW creates includes several nodes by default. Of primary interest are the nodes for the **User Parameter** and **Event Data**. The latter can be used to access arguments passed from the ActiveX control. Below is a complete block diagram for a callback VI that processes PCIEvent1w.



In the example above the value of the PMC408's interrupt status register is passed from the ActiveX control to the Callback VI. The VI unbundles the status value from the Event data node and writes it to the Value property of the Status indicator (found on the front panel of the main VI). Note that if the callback VI handles an event that does not pass any arguments, the Event data node will not be present.

Distribution Files

PCI Win32 Driver Software (PCISW-API-WIN) consists of three primary sets of components:

- The kernel-mode driver
- A suite of Windows 32 Dynamic Link Libraries (DLLs)
- An event notification ActiveX control

Kernel-mode driver

Drvxwdm.sys exports hardware control services from the kernel. Applications communicate indirectly with the driver through the functions exported from the DLLs.

Windows 32 DLLs

Each Acromag PCI module has a corresponding Windows 32 DLL that provides the Application Programming Interface (API) used to access the card. DLL filenames are in the form PCIXXX.dll, where "XXX" refers to the PCI module's model number. Each DLL is written in C and contains functions using the _stdcall calling convention. For each DLL there is a corresponding Function Reference document that describes the functions provided by the DLL in detail.

PCI Event ActiveX Control

The PCI Event ActiveX control (PCIEvent.ocx) may be used for interrupt notification and processing in environments that do not support callback functions (LabVIEW) or where there are complications implementing thread safe code. In addition to the OCX file there is also a type library file (PCIEvent.tlb). Some development environments use this file to obtain information about the control's methods and events.

Redistribution Requirements

When developing an application that utilizes the driver software, the following files must be installed on the target machine.

Windows 2000 Files

- Your application program
- All DLL's corresponding to the PCI modules you are using. These are typically installed in your application's directory.
- The Microsoft® C Runtime Library (msvcr71.dll). This file is typically installed in your application's directory.
- In the \winnt\system32\drivers directory: Drvxwdm.sys

Windows 7, Vista and XP Files

- Your application program
- All DLL's corresponding to the PCI modules you are using. These are typically installed in your application's directory.
- The Microsoft® C Runtime Library (msvcr71.dll). This file is typically installed in your application's directory.
- In the \windows\system32\drivers directory: Drvxwdm.sys

PCI Event Control files

If you are using the PCI Event ActiveX control the following files are needed in addition to those listed above.

- PCIEvent.ocx, PCIEvent.tlb. The ActiveX control needs to be registered on the system using the Regsvr32 tool. Regsvr32.exe is included with Windows and is installed in the System directory (e.g., C:\Windows\system32).
- MFCDLL Shared Library (mfc71.dll). This file is typically installed in your application's directory.

DLL Location Notes

To reduce the likelihood of "DLL Conflict" issues Microsoft recommends that DLLs be installed to the application directory with the program executable. This is the preferred location when running a single executable. However, if several applications will be simultaneously sharing a PCI DLL it is recommended that the DLL be placed in a common directory. This allows the shared DLL to properly track which boards are in use.

In order for the operating system to find a DLL, its location must be part of the Windows search order. The typical search order is as follows:

1. The directory from which the application is loaded
2. The system directory (e.g., C:\Windows\system32)
3. The 16-bit system directory (e.g., C:\Windows\system)
4. The Windows directory (e.g., C\Windows)
5. The current directory
6. The directories listed in the PATH environment variable

The easiest solution to sharing a DLL is to place it in the Windows or Windows system directory. However, many applications store DLLs in these directories so using these locations creates the most risk for DLL conflict issues.

The technique used by the PCISW-API-WIN installer is to append the common DLL directory (typically C:\Acromag\PCISW_API_WIN\redist) to the PATH environment variable. This allows the appropriate DLL to be located when running each example project.

MODIFYING THE PATH SETTING

Use the following steps if you wish to modify the PATH setting on a target machine.

Windows 7, Vista, XP and 2000

1. Launch the System applet from the Windows Control Panel.
2. Select the Advanced link (or tab) and then click the Environment Variables button.
3. Locate "Path" in the User Variables or System Variables. The PATH is a series of one or more directories separated by semicolons.
4. Edit the variable by appending the path to the common DLL directory to the right of the existing value.
5. Click OK